Lattice and Microsoft C Users' Group

Issue No. 13 **Published July 20, 1987** **Nominally May / June 1987**

Books on User Interface Design

My training as a scientist forces me to check the published literature to see what's already been done before I plunge into designing an algorithm or a user interface. The price I pay is reading a lot of mediocre books and articles for every rewarding find. But this month I found a real winner - more than adequate compensation for this month's dud. This article covers some books that discuss user interface design as a research topic, as a craft for working programmers and as an art form. The topic is relevant to practical programming work but much of what's in print lags behind what today's best PC programs are already doing.

Designing the User Interface by Ben Shneiderman. (Addison–Wesley, 1987.) This is a survey of research in user interface design by a computer science professor. Many research papers are cited and each chapter ends with several pages of references. But most of the discussions are too short and too shallow to be useful. Many of the experiments discussed are too simple-minded to be relevant. The section on response time research is detailed enough but the research cited is poorly designed. Shneiderman is an authority in this field so I expected this book to be good but I found it to be painfully dull. Academic stuff and not relevant to the real world.

User Centered System Design, edited by Donald A. Norman and Stephen W. Draper. (Lawrence Erlbaum Associates, 1986. ISBN 0- 89859-872-9. $20.75) The editors assigned topics to many researchers but they worked the result into a useful collection of papers. There is too much jargon introduced and too many conceptual models built but several papers gave me new viewpoints for thinking about user interface design. The papers "Direct Manipulation Interfaces", "Notes on the Future of Programming: Breaking the Utility Barrier" and several chapters on user understanding were all useful. Still academic stuff but useful to stimulate your thinking.

The Human Factor, Designing Computer Systems for People by Richard Rubinstein and Harry Hersh with Harry Ledgard. (Digital Press, 1984. $25.00). The authors are all Ph.D.s but this is real world stuff – relevant and meaty. Concepts and ideas are combined with immediately useful specific suggestions. Most sections of 1/2 to 2 pages are summarized in a short one sentence guideline. In 250 pages, the book discusses testing, documentation and overall system design as well as the user interface itself. Chapters on the user's conceptual model and responding to users are especially well-done and useful. I highly recommend this book for any programmer.

The Elements of Friendly Software Design, Paul Heckel. (Warner Books, 1984. $ 8.95). Heckel discusses software as communication medium and art form rather than as engineering. His examples are interesting and thought provoking. For example, he points out while Thomas Edison pioneered the technology of movie-making, he didn't understand it as a communication medium. D.W. Griffith developed the art of movie-making. There is too much pop culture here but there are also unusual ideas.

Computer Literacy (520 Lawrence Expressway, Sunnyvale, CA 94086. Phone (408) 730-9955.) should have most of these books. They handle mail and phone orders. The Norman and Draper book may be hard to find; I got a copy at the Cal Tech bookstore in Pasadena, California.

Keyboard Input for Interactive Programs

The ANSI standard provides several functions for console input - `getchar`, `gets` and `scanf` are the main ones. But the standard does not specify whether input is buffered, how control characters and special keys are handled and other important details. This lack of detail is understandable since C must be implementable on computer systems with a wide variety of console I/O facilities. But as a result, standard library functions are unsatisfactory for console input in highly interactive programs. In this article I explore alternatives for console input in PC- DOS C programs.

In Lattice C, Microsoft C, and other MS-DOS C compilers I have used, these standard library functions provide line buffering, automatic echoing of typed characters and suppression of function key and other special key input. This is a good choice for casual programs but it is not appropriate for serious interactive programs. Instead, console input functions should have the following characteristics:

1. Immediate Access to single keystrokes - An interactive program should sense and respond to a keystroke immediately. Programs that must wait for an entire line of input to be typed before receiving any characters cannot provide an effective user interface.

2. Access to all PC keys - control keys, function keys, cursor control keys and other special keys must be passed to the application without being translated or suppressed.

3. No Automatic Echoing - The application program must control echoing of keyboard input so that it is correct in the context of the executing application program. If keyboard input is automatically echoed by library functions or by DOS services, it may compromise user interface design.

4. No Side Effects - Some DOS calls check for special characters and trigger side effects: Control-Break and Control-C cause an INT 23H interrupt and may abort program execution. Control-PrtSc and Control-P toggle echoing of console output to the printer and Control-S and Control-NumLock freeze console output until another character is typed. Control-Z signals an End Of File condition for console input.

5. Status Checking Without Waiting - Some interactive programs need to check for the presence of keyboard input without waiting if no input is available.

You can alter the behavior of library functions such as `getchar` to meet some of these criteria. The DOS IOCTL call can be used to set binary mode for DOS file handle 0. (This disables DOS checking for Control-C and other special characters. The PC-DOS 3.x Technical Reference manual discusses the effects of binary mode versus ASCII mode on pages 4-9 through 4-12.) The C library functions ordinarily operate in TEXT mode for console input (file handle 0.) The makebin functions in issue #11, the setmode function for Microsoft and Turbo C and iomode for Lattice C change file handle 0 to BINARY mode The `setbuf` and `setvbuf` functions can defeat line buffering for the `stdin` stream. All this relies on poorly documented implementation details of library function – not a solid basis for building an application.

The `getch` function in the Lattice, Microsoft and Turbo C libraries is not defined by the ANSI standard but it does meet most of the criteria discussed above. It gives immediate access to single keystrokes including control keys, function keys and cursor control keys. The companion status function, `kbhit`, checks for the presence of keyboard input without waiting for input. The Turbo C version bypasses DOS side effects on Control-C and other characters. The Lattice and Microsoft versions do trigger DOS Control-C checking and other side effects so they are not completely satisfactory.

Even if the `getch` function doesn't trigger side effects itself, such checking may still cause problems with getch. DOS also performs Control-C checking during console output and other calls. Due to a bug in DOS handling for PC extended keyboard codes, your program may hang up waiting for a character which will never appear. PC keys which do not correspond to an ASCII character are translated into a two character sequence by the PC-DOS console input device driver as the examples below show:

Key	DOS returns	Mistaken for
F1 Function key	00 then 59	
Right Arrow key	00 then 77	
Alt-Q	00 then 16	00, Control-P
Alt-R	00 then 19	00, Control-S
Control-@	00 then 03	00, Control-C

The second character is the scan code assigned by the PC BIOS. When getch returns a zero value, this indicates that a scan code follows in the next character. So a C program getting a zero return value from getch should immediately call getch again to collect the scan code. However, MS-DOS performs special character checking on the scan codes - this is nonsense but DOS does it. Thus Alt-Q is triggers the response for Control-P, Alt- R triggers the action for Control-S and Control-@ triggers the Control-C action. The scan code itself is absorbed by DOS. A program using `getch` will receive the 00 value but the scan code will not appear. getch will wait until another character is typed. If you do use `getch`, you should check for status and exit after a timed interval in case an expected scan code is absorbed by DOS.

The simplest solution to these problems with DOS based console input is to use the PC BIOS keyboard input services (INT 16H) directly. This service meets all the criteria defined above. The keyiob.c source file below uses the Lattice int86 library function to implement the `getkey` function which returns a single keystroke value. `getkey` waits for input if none is available; the keypress function checks for waiting keyboard and returns immediately.

The Microsoft and Turbo C `int86` library functions do not give access to the zero flag so the keypress function can't be implemented with `int86` for those compilers. The keyiobm.asm file provides an assembler language alternative. The LMCUG.MAC file and the DOSM.MAC file it uses were described in issue #12 and are on the LMCUG BBS.

The BIOS also provides a call to check the state of the PC shift, control and alt keys - depressed or not. The same call returns the state of the Insert, Caps Lock, Num Lock and Scroll Lock toggles. The keyiob2.c file uses int86 to implement this call. The shift key status bits are

Insert State	0x80	Alt key Depressed	0x08
Caps Lock State	0x40	Ctl key depressed	0x04
Num Lock State	0x20	Left Shift depressed	0x02
Scroll Lock State	0x10	Right Shift depressed	0x01

A single call to `getkey` returns one keystroke - a simpler and more satisfactory approach than the two characters per special key scheme that DOS uses. Keystrokes for which the ASCII char is zero are returned as 0x100 plus the scan code. A similar `getkey` function based on the Lattice/Microsoft `getch` function was presented in issue #2 along with a keys.h header file which defined constants for some special keys. The IBM XT and AT Technical Reference Manuals and Peter Norton's book Programmer's Guide to the IBM PC (Microsoft Press. 1985.) list PC scan codes and discuss how the PC BIOS keyboard support works.

```c
/* keyiob.c - BIOS based getkey() and keypress() */
/* works for Lattice C 2.15 on.  NOT for MSC or Turbo C */
#include "dos.h"

#define KEYIO_CALL 0x16        /* BIOS keyboard interrupt */
#define CHK_STAT   0x01        /* check status function code */
#define GET_CHR    0x00        /* get character function code */
#define Z_FLAG     0x40        /* 8086 flag 1= zero result */

int keypress()                 /* is a keystroke waiting? */
 {                             /* return - 1=yes , 0=no */
   union REGS r ;
   int stat ;

   r.h.ah = CHK_STAT ;         /* fun. code = check status */
   stat = int86(KEYIO_CALL,& r,& r) ;
   if( (stat & Z_FLAG) == 0 )
      return( 1 ) ;            /* input waiting */
   else return( 0 ) ;          /* no input waiting */
}


int getkey()                   /* get a single keystroke */
 {
   union REGS r ;
   int c ;

   r.h.ah = GET_CHR ;
   int86(KEYIO_CALL,& r,& r) ;
   c = r.h.al;                 /* get ASCII char value */
   if( c == 0 )                /* if not an ASCII char */
      c = 0x100 + (r.h.ah & 0xff) ; /* use scan code + 256 */
   return( c ) ;
 }
```

```asm
;          BIOS based getkey() and keypress() functions

           include lmcug.mac
KEYIO_CALL EQU  16H             ; s/w interrupt number
CHK_STAT   EQU  1               ; check status function code
GET_CHR    EQU  0               ; get keystroke function code

           PSEG
           CFUN    keypress     ; keypress() function
           MOV     AH,CHK_STAT
           INT     KEYIO_CALL   ; make BIOS call
           MOV     AX,0         ; return zero if zero flag set
           JZ      out
           MOV     AX,1         ; return one if zero flag not set
out:
           RET
           CFEND   keypress

           CFUN    getkey       ; getkey() function
           MOV     AH,GET_CHR
           INT     KEYIO_CALL
           OR      AL,AL
           JNE     ascii        ; is it an ASCII char ?
           MOV     AL,AH        ;   no - return scan code
           MOV     AH,1         ;          + 0x100
```

```
        JMP     key2
ascii:
        MOV     AH,0            ;    yes - return ASCII code alone
key2:
        RET
        CFEND   getkey

        ENDPS
        END
```

```c
/* keyiob2.c - BIOS based shift key status */
/* works for Lattice C, MSC or Turbo C */
#include "dos.h"

#define KEYIO_CALL 0x16      /* BIOS keyboard interrupt */
#define SHFT_STAT  0x02      /* check shift key status fun. code */

int shiftkey()               /* get shift key status */
 {                           /* return BIOS defined byte */
   union REGS r ;

   r.h.ah = SHFT_STAT ;      /* fun. code = get shift status */
   int86(KEYIO_CALL,& r,& r) ;
   return( r.h.al & 0xff );/* return keyboard flags byte */
 }
```

Collecting a Line of Console Input

The previous article provided a functions for single keystroke console input. When several characters are required, we need to echo each character as it is typed and collect characters until the end of the field or line is reached. The C library functions, `gets` and `fgets`, almost fit the description. They collect and echo characters until a carriage return is typed. But these functions have the disadvantages described in the previous article. Since they suppress many Function key and ALT- Key keys and don't return unless the return key is pressed, we can't design the user interface to be sensible and slick. If we use `gets` or `fgets`, we have to make the user interface fit the limits of those functions.

As usual for C, the solution is to write a functional replacement for `gets` and `fgets`. The `xgets` function shown below echoes printable ASCII chars and collects them in C string form. But it returns immediately when any other key is pressed - the return value identifies the key pressed. That allows the application program to tailor the use of control keys, function keys and cursor keys. Since line editing facilities are usually required, `xgets` provides for single character deletion and deletion of the whole string. The keystrokes for those services are set as backspace (`0x08`) and Control-X (`0x18`) but they can be changed by the application program. `xgets` uses the `xs_putc` and `xs_erase` functions in xs_out.c to echo a printable character and to erase previously echoed characters.

In some applications, you may wish to accept a string immediately when the string is filled - before a return or other non- printable character is typed. `xgets` returns immediately with a when the string size count is reduced to zero. In the getfld.c file below, `xgets` is be called again with a zero string length to wait for a return or other terminating key to be pressed. The getfld.c file also shows how special keys can be ignored. The Tab, return and the F1 function key all terminate the field, so getfld returns when they are typed. `getfld` ignores other keys and calls xgets again to continue collecting keystrokes.

The `xgets` function provides a low-level building block for simple console input. The xs_out.c file can be replaced with functions that use BIOS output, direct screen writing or calls to a windows library. `xgets` can be used for full-screen input where a number of prompts and input areas are defined. As written, the function doesn't use screen attributes nor does it work in insertion mode. Displaying prompts and positioning the cursor are also left to other functions. `xgets` it does provide a useful starting point and it illustrates how line oriented input can be combined with intelligent handling of the full IBM PC keyboard.

```
/* xgets.c - collect & echo chars until non-printable char received */
/* put chars in C string format */
#include "stdio.h"
#include "ctype.h"

/* line editing chars - you can change these */
/* or set them to -1 if no editing desired */
int bs_char = 0x08 ;              /* char for single backspace */
int cs_char = 0x18 ;              /* char to clear entire string */
                                  /* backspaces to start of string */

int xgets(s,smax,start)           /* get char string from keybd & echo */
 char *s ;                        /* start putting chars here */
 int smax ;                       /* maximum no. chars to allow */
 char *start ;                    /* don't backspace beyond this point */
 {                                /* return last char typed */
   int c ;

   while( 1 )
      { c = getkey() ;            /* get next keystroke */
        if( c == bs_char )
           { if( s != start )/* anything to delete ? */
                { xs_erase(1) ; /*  yes - delete one char */
                  s-- ;
                  smax++ ;
                }
           }
        else if( c == cs_char)/* line deletion char ? */
           { if( s != start )   /*  yes - anything to delete? */
                { xs_erase(s - start) ; /*  yes - delete all */
                  smax = smax + (s-start) ;
                  s = start ;
                }
           }
                                  /* printable char ? */
        else if( (c >= ' ') && (c <= '~') )
           { if( smax < 0 )     /*   room for it ? */
                break ;         /*     no - exit */
             xs_putc(c) ;       /*   echo it */
             *s = c ;           /*   store it in string */
             s++ ;              /*   advance string pointer */
             smax -- ;          /*   reduce field max. size */
             if( smax == 0 )    /*   if this char filled string, */
               { c = -1 ;       /*     return special value */
                 break ;
               }
           }
         else break ;         /* non-printable char - not BS or CS */
      }
   *s = '\0' ;                /* terminate the string */
   return( c ) ;              /* return the last char typed */
 }
```

```
/* xs_out.c - console output support for xgets() */
#include "stdio.h"

int xs_putc(c)                          /* output one char to the screen */
 int c ;
 {
   putchar(c) ;
 }


xs_erase(n)                             /* erase previous screen positions */
 int n ;                                /* number of positions to erase */
 {                                      /* assumes everything on current line */
   int i ;

   for(i=0;i<n;i++)                     /* back up n positions */
     { putchar('\b') ; }
   for(i=0;i<n;i++)                     /* write blanks */
     { putchar(' ') ; }
   for(i=0;i<n;i++)                     /* back up again */
     { putchar('\b') ; }
 }


/* getfld.c - collect input from a field */
#include "stdio.h"
#define F1_KEY  (0x100 + 59)
#define BEEP     0x07

int getfld(fs,fsmax)
 char fs[] ;                            /* put the field here */
 int fsmax ;                            /* maximum number of chars */
 {
   int ret ;                            /* terminating key */
   char *next ;                         /* current position in field */
   int flen ;
   int fmax ;

   next = fs ;
   fmax = fsmax ;
   while( 1 )
       { ret = xgets(next,fmax,fs) ; /* collect chars */
         flen = strlen(fs);       /* find end of string so far */
         if( (ret == '\r')        /* Carriage Return, Tab or F1 ? */
            || (ret == '\t') || (ret == F1_KEY) )
               break ;            /*   yes - thru */
         else if( ret == -1)      /* filled field ? */
            { ; }                 /*   yes - wait for proper termination */
         else if(isprint(ret))    /* typed past the end of the field? */
            { putchar(BEEP);      /*   yes - warn user */
            }                     /*         wait for proper termination */
          /* ignore other chars and call xgets again */
         next = fs + flen ;       /* start at end of string */
         fmax = fsmax - flen ;
       }
   return( ret ) ;
 }
```

Validating Numeric Input

The C library provides several functions for converting an ASCII character string into a number: the `sscanf` function handles a number of data types and the `atoi`, `atol` and `atof` functions are specific to a single data type. None of these functions provides satisfactory validation of the input character string. They all stop when an invalid character is found and return the value of the number accumulated to that point. The examples below show how some invalid input is handled by the `atol` function:

Input String	returned by `atol`
"x"	0
"12x34"	12
" - 1"	0
"12 34"	12

In practice, these C library functions are not usable in serious programs. Fortunately, we can easily write equivalent functions that do provide adequate validation. The `vatol` function in vatol.c shows how validation can be added to the atol function. Extracting numeric characters stops when a non-numeric character is found. If the input is valid, the conversion should have reached the end of the input string.

`vatol` returns a special value, `INV_LNO`, when an error is detected. The validate.h file defines this constant and others. The `vatol` input argument, `mode`, specifies whether leading spaces, a sign or trailing spaces will be allowed. When an error is detected, the current position in the string is recorded in the pointer variable `pverr`. `pverr` is declared as `extern` in the validate.h header file so that it is shared by all validation functions and accessible in the application functions that use call them. The pverr.c file provides a single place to declare `pverr` as a global variable rather than as extern.

There are a few errors that `vatol` doesn't not detect. It accepts a null string and returns a zero value. Strings with only blanks are accepted as are strings with a plus or minus sign and no digits. It does not check the size of the number so the value may overflow. (I decided to place such checking outside vatol.)

In writing this article, I developed a companion function, `vatod`, for string to double conversion. That function and test programs for both functions will be available on the LMCUG BBS system. Data Handling Utilities in C by R. A. Radcliffe and T. J. Rabb (Sybex. 1986) gives a complete and rather complex validation function. The C Library by Kris Jamsa (Osborne McGraw-Hill. 1985) provides functions similar to vatol and vatod. K&R lists simple atoi and atof functions without validation.

```
/* validate.h - constants for input validation */
                                /* modes for numeric input */
#define LEADING_SPACES  1
#define TRAILING_SPACES 2
#define SIGN_ALLOWED    4

#define INV_LNO 0x7fffffff /* invalid return for vatol() */
#define INV_FNO   1.999E30 /* invalid return for vatod() */
extern char *pverr ;       /* record postion of invalid char here */
```

```
/* pverr.c - define and allocate space for pverr variable */
/* used in vatol() and valod() */
char *pverr ;
```

```
/* vatol.c - ASCII string to long conversion with validation */
#include "validate.h"
#include "ctype.h"

long vatol(s,mode)
 char *s ;                    /* string containing number */
 int mode ;                   /* what to allow */
 {
   long n ;
   int sign ;

   n = 0L ;
   sign = 1 ;
   if( (mode & LEADING_SPACES) != 0) /* remove leading blanks ? */
      { while( *s == ' ')
          { s++ ; }
      }

   if( (mode & SIGN_ALLOWED) != 0)/* look for sign ? */
      { if( *s == '+')
          s++ ;
        else if( *s == '-')
          { sign = -1 ;
            s++ ;
          }
      }
                              /* get digits and place in number */
   while( (isdigit(*s) != 0) ) /* stop at non-numeric char */
      { n = n*10 + *s - '0' ;
        s++ ;
      }

   if( (mode & TRAILING_SPACES) != 0) /* move past trailing blanks ? */
      { while( *s == ' ')
          { s++ ; }
      }
   if( *s != '\0')              /* reached the end of the string */
      { pverr = s ;             /*   no - mark the location */
                                /*        of the invalid char */
        return( INV_LNO ); /*        and return error value */
      }
   if( sign == -1 )             /* check for negative sign */
      n = - n ;                 /*   yes - make value negative */
   return( n ) ;                /* return value of the number */
 }
```

BBS and Issue Dates

The LMCUG BBS is up and running 24 hours a day; the phone number is (415) 935-7275 and 300 and 1200 baud are supported. You can sign yourself up the first time you call the BBS; I have been unable to pre-register members names. For the present, access to the BBS is free to LMCUG members - 45 minutes per day. BBS access will continue to unlimited until I replace the current BBS software. I am hoping that your contributions – files and public messages – will make the BBS system valuable to all members.

If your address label says 5/87, this is your last issue. If you are a new member, don't be alarmed by the nominal issue dates – they are always out of sync. The mailing label specifies the last issue you will receive. Since that issue will be out of sync, you will get a full year's membership.

Turbo C - A Step Forward for C Compilers

I have been using Turbo C for several weeks now and I think it is a fine product. I have found no bugs and had no problems. Borland has set a new standard for value. For a list price of $ 99.95 or a street price of $ 60-65, you can buy a C compiler without compromises and with plenty of bells and whistles. From now on, a $ 250 product will have to offer more than just a good compiler and a standard C library.

The compiler's speed is a standard setter, too. On my 8 MHZ AT clone, compile ranged from under 2 seconds (for 60 lines) up to 6 seconds (for 500 lines). Times for compile and linking those files were 7 seconds and 9 seconds. This does more than save a few seconds - it changes the rhythm of editing, compiling, linking and execution. Since you type much less and wait only a few seconds at each step, you can concentrate continuously on writing and debugging C source code.

Turbo C's integrated development environment makes the compiler's speed genuinely useful. Pull-down menus and keyboard shortcuts (ALT and Function keys) eliminate the need to remember lots of commands. You enter and edit source files with a full screen editor like that in Turbo Pascal and other Borland products. When you finish editing a source file, pressing a single key compiles it. If syntax errors are detected, pressing one key produces a display of error messages in one window. The source file is displayed in another window with the line in error highlighted. A single keystroke re-enters the editor with the cursor positioned on that line. You can fix a single error and re-compile the file or scroll through the error messages and fix each error before re-compiling.

Syntax error detection and recovery is better than that of Lattice 3.2 or Microsoft 4.0 (they produce an avalanche of spurious messages.) but worse than that of the DeSmet, Eco-C or Datalight compilers. Overall, quick compilation with a single keystroke makes correcting syntax errors fast and painless.

For programs with more than one source file, Turbo C uses a project file (a text file with a .PRJ extension) listing the source files that make up the program. When you press the F9 key, Turbo C re-compiles any files that have been changed and re-links the program. For small projects and debugging, creating a .PRJ file it is much easier than using Microsoft's MAKE program. Correcting syntax errors in multiple source files requires some extra work –Turbo C doesn't load the appropriate source file into the editor when a syntax error is encountered; you have to load it yourself.

The Turbo C compiler can also be invoked from DOS with the TCC command. This command works much like the Lattice LC and Microsoft CL commands. You can compile and link one or a number of C files. There are many command line options for memory models, optimization levels, levels of warning messages and many other things. Turbo C is a full function C compiler with six memory models, support for function prototypes and other ANSI features, and a library more comprehensive than that of Lattice 3.2 or Microsoft 4.0. It interfaces with .OBJ files produced with the standard MASM macro assembler and works with the .LIB files produced with the Microsoft LIB program. A full-featured standalone MAKE program is provided as is the Turbo linker.

Turbo C also supports the NEAR, FAR and HUGE keywords that Microsoft C pioneered. It provides extensive features for inline ASM code; the documentation is almost adequate to explain these features. (That is better than other compiler vendors such as MANX or C Ware provide.) The `interrupt` keyword allows C functions to be used directly as interrupt handlers. The library contains useful functions for critical error handling, absolute sector disk I/O and many other IBM PC specific functions.

As Table 1 shows, Turbo C compiles source files much faster than Lattice 3.2 or Microsoft 4.0. Execution times are faster than those for Lattice 3.2 in the small model and about the same for the large model. Against Microsoft 4.0, Turbo C's execution times have some victories and some losses. Any of these compilers is adequate for producing quality programs; none will turn a frog into a prince. Differences in .EXE file size reflect library organization – not important in sizable, real-world programs.

Library source code is available for an additional $ 295 from Borland. If you are not building a serious in-house program or a product, you can live without it. If you need source code, Turbo C is really a $ 360 purchase.

Turbo C documentation sets new standards but it has some weak points too. The manuals do more than document compiler options and library functions; they contain 60 pages of C language and library introductory discussions, 16 pages for Turbo Pascal programmers and an informal 30 C language reference chapter. This material will help newcomers to C. A discussions of C pitfalls is also relevant as is a long discussion of the scanf function. But descriptions of most functions are too short and there are no topic discussions for file I/O, memory management or the DOS interface. Both Lattice and Microsoft manuals are better in this respect. Installing Turbo C and learning to use it is not difficult but the manuals misplace some important information.

The weakest point about Turbo C is the editor. It is limited to a single window and a single file. Tabs are fixed at 8 character intervals and the WordStar-like commands are hard to remember. (For example, Control-K followed by B to mark the beginning of a block and Control-K K to mark the end.) PC keypad keys for cursor movement (arrow, Home, PgUp) are used and a configuration program allows commands to be assigned to other keystrokes. However, many ALT and function keys are reserved for Turbo Hot keys. Compared to modern editors like BRIEF and VEDIT, the Turbo C editor is limited and inflexible. I like Turbo C in spite of this antique editor, not because of it.

Turbo C is the first C compiler to be tailored for enthusiasts and students. The integrated environment is easy to master and the documentation helps a novice started writing C programs. The compiler's speed makes the process of entering, compiling and debugging simple programs fun. Turbo C is useful for serious projects too; the main advantage over Lattice or Microsoft is the speed and ease of compiling source files and fixing syntax errors.

Quick C – Microsoft Strikes back

Perhaps in response to Turbo C, Microsoft has announced that Quick C and version 5.0 of Microsoft C will be available in September. I have used beta test versions for a couple of weeks. Since neither product is in a finished state, this is just a preview. If the finished products deliver what the beta test versions promise, Microsoft will be setting the pace.

Quick C offers an integrated full-screen editor, compiler and a make capability like that of Turbo C at a similar $ 99 list price. I found Quick C compile times to be slightly slower than Turbo C's when producing .OBJ files. Quick C can also compile and link programs directly into memory; those times are better than Turbo C compile and link times. When an .EXE file is needed, Quick C invokes the MS-DOS linker (LINK) which requires 5-10 seconds more than for Turbo C. Because of a bug, I could collect compile or compile and link times for multiple source files. Execution times are not shown because Microsoft has not finished work on optimization in Quick C. (Current results are a bit slow but not that bad.)

Quick C integrates the CodeView source level debugger into the development environment - Turbo C has no debugger. While you are debugging, source code is displayed on the screen. Editing changes can be made directly at any time. After editing changes are made, re-compilation and re-linking are done automatically when you execute a single step or continue execution command. Quick C couples editing, compiling and debugging in a smooth, slick environment like a good C interpreter. Quick C needs lots of memory – over 300K of RAM memory. The debugger probably won't be usable on large programs but that is inevitable with the current 640K limit of PC-DOS.

Quick C is included in version 5.0 of Microsoft C. Upgrades to version 5.0 for MSC 4.0 owners will be $ 50 and will include Quick C. This is a terrific deal! As table 1 shows, MSC 5.0 gives significantly better execution times. Microsoft found improvements of 10 to 15 % on recompiling their applications.

Library source code will be available for $ 150. Quick C and MSC 5.0 share the same libraries including a new library for bit-mapped screen graphics. Many MS-DOS and IBM PC specific library functions have also been added; the MSC 5.0 library seems competitive with the Lattice and with Turbo libraries.

Table 1
Compiler Benchmark Results

	Turbo C 1.0	Lattice 3.2	MSC 4.0	Quick C beta	MSC 5.0 beta
Compile Times in Secs. - (Compile & Link Times in Parentheses)					
1 file					
60 lines	2 (7)	7 (15)	11 (16)	2 (10)	12 (17)
150 lines	3 (8)	13 (22)	19 (29)	4 (16)	20 (27)
500 lines	6 (9)	25 (34)	28 (35)	6 (18)	29 (33)
6 files					
150 lines	- (14)	33 (44)	56 (67)	-	57 (67)
Execution times in Secs. for Small model - (large model in parentheses)					
Sieve	37 (37)	50 (50)	59 (59)	-	31 (31)
with reg. var	23 (24)	24 (24)	22 (22)	-	16 (16)
String Copy	52 (109)	114 (146)	96 (109)	-	38 (71)
with reg. var	26 (56)	57 (73)	20 (55)	-	19 (36)
Char Count	61 (74)	77 (95)	53 (59)	-	36 (52)
int arithmetic	53 (53)	71 (71)	50 (50)	-	39 (39)
long arith.	213 (214)	215 (224)	202 (214)	-	175 (182)
float add/mult	17 (17)	17 (18)	13 (13)	-	10 (10)
float exp/log	54 (55)	130 (142)	109 (109)	-	103 (113)
.EXE file size (Kbytes)					
60 lines (no float)	8 K	10 K	9 K	11 K	10 K
150 lines (float)	23 K	18 K	24 K	24 K	26 K
500 lines (no float)	11 K	14 K	14 K	15 K	15 K

Notes
1. All tests run on 8 MHZ AT clone w/o 80287 floating point chip.
2. Entries with - mark tests that were not or could not be run.
3. Function call and read/write benchmarks gave similar results for all compilers.
4. Benchmarks from "State of C", PC Tech Journal, Jan. 1986.

Using Far Pointers in C Programs

The C language was developed on computers with a flat address space. Compiler vendors have worked around the Intel 8086 segmented architecture adequately, but one result is that MS-DOS C compilers offer a number of memory models. It is up to you to balance the execution speed of 16 bit pointers against the access to more than 64 Kbytes that 32 bit pointers allow. Declaring individual pointers with the `far` keyword allows you to get the benefits of both alternatives. The `far` keyword isn't ANSI standard but Microsoft, Turbo C and now Lattice v3.2 all support it. The `far` keyword can be used in several ways as the compiler manuals illustrate. Those discussions are pretty complex; this article shows some real, practical uses of far pointers.

```
char far *pcfar ;          /* pointer to char */
                           /* 32 bits - regardless of memory model */
char far *pifar ;          /* pointer to int */
                           /* 32 bits - regardless of memory model */

c = *pcfar ;               /* using a far pointer - looks the same */
*pcfar = c + 1 ;           /* as a normal pointer */
strcpy(s,pcfar) ;          /* won't work unless the memory model */
                           /* provides 32 bit pointers itself */
```

Memory Model Independent Access to fixed memory locations - programs sometimes need access to memory locations outside the program's data area - to the display adapter memory for example. In a memory model with 32 bit pointers this is easy - a pointer reference like `*p` will do. But 16 bit pointers (in the small model for example) can only address the program's data area - 64 Kbytes maximum. Library functions such as the Lattice peek and poke functions provide access to all of memory but they slow and not standard. Far pointers allow efficient use of 32 bit pointers in any memory model.

In the testfar.c program below, the `pcrt_mode` pointer is used to retrieve the current mode for the display adapter. The starting address of the screen is then set to `0xB000:0000` (monochrome adapter) or `0xB800:0000` (CGA or EGA adapter). The `for` loop then writes 400 character and attribute pairs to the screen - advancing the screen address after each write.

```
/* testfar.c - use far ptrs for direct writes to screen buffer */
/* assumes that adapter is in a 40 or 80 column text mode */
#include "stdio.h"
#define MONO_MODE 7

main()
 {
   char far *pcrt_mode ;
   unsigned far *pscn ;
   int i ;

   pcrt_mode= (char far *) 0x00400049 ;/* CRT mode stored here */
   if( *pcrt_mode == MONO_MODE )        /* set starting screen address */
        pscn = (unsigned far *) 0xB0000000 ;
   else pscn = (unsigned far *) 0xB8000000 ;
   printf(" screen mode & address: %x  %p \n",*pcrt_mode, pscn );
   getchar() ;
   for(i=0; i<400 ; i++)                /* write 5 rows of chars & attrib. */
       {    /*  char value      attribute value */
        *pscn = (i & 0xff) | ( (i & 0xf0) << 8 ) ;
        pscn++ ;
       }
 }
```

This example demonstrates two uses of far pointers: direct screen memory access and access to the RAM parameter area maintained by the PC BIOS. Getting and setting BIOS parameters by BIOS software interrupts is preferable to direct access to the parameter area but some parameters are not accessible via software interrupts. The IBM PC Technical Reference Manual lists the contents of the BIOS parameter area which starts at address `40:0`. The list below gives some of those parameters and their addresses:

Starting Port Addresses		
RS 232 adapter	40:0	1 word per printer (4)
Printers	40:8	1 word per printer (4)
RESET FLAG	40:72	if 1234H, bypass memory test
(warm or cold re-boot)		
Timeout Values		
Printers	40:78	1 byte per printer (4)
RS 232 adapter	40:7C	1 byte per adapter (4)

Using more than 64K bytes of data in the small memory model - Some programs require access to more than 64 Kbytes of data for a single use. For example, a file manager program might benefit from a single large buffer pool. Such programs can use the small or medium or program memory model with efficient 16 bit pointers. The 32 bit far pointers can be limited to the places they are really needed. Microsoft C provides the `_fmalloc` and `_ffree` functions that may allocate and free all of available memory - not just a total of 64 Kbytes. The code fragment below shows how `_fmalloc` and `_ffree` are used. Note that far pointers can't be passed as arguments to library functions such as `fgets` and `strcpy` when a memory model with 16 bit pointers is being used.

```
char far *_fmalloc(unsigned );           /* get memory from far heap */
void _ffree( char far *) ;               /* return memory to far heap */
char s[201 ] ;    char *p ;
char far *pfar ;  char far *pf2 ;
...
     fgets(s,200,in) ;                   /* get a string into data area */
     p = s ;
                                         /* allocate memory from far heap */
     pfar = (char far *) _fmalloc( strlen(s)+1 ) ;
     while( *p != '\0' )                 /* copy to far area */
        { *pf2++ = *p++ ; }
...
     _ffree(pfar) ;                      /* frees memory allocated to far heap */
```

Writing Device Driver Programs - C programs to be used as MS-DOS device drivers must be position independent; DOS must be able to load them anywhere in memory with no relocation of segment addresses. This requires position independent programs; all function and data addresses embedded in the .EXE file must be 16 bit values rather than 32 bit values. The small memory model fulfills these requirements but device drivers often need access to more than 64K bytes of data. Far pointers can provide this access as long as they are not initialized in the C source files.

```
     char far *pscn ;
     char far *pc ;
     char p2 ;

     pscn = (char far *) 0xB0000000 ;  /* OK */
     pc = "abc" ;              /* NOT OK - requires relocation */
     p2 = "def" ;              /* OK for small model */
```

Letters

Linking BASIC with C - Susan Scott (The CBORD Group, Inc., 202 E. State Street, Suite 300, Ithaca, NY 14850-9990) asks about linking BASIC object modules with C programs. Your chances of success are better if the compiler vendor supports mixed C and BASIC programs. Microsoft's Quick-C and Microsoft C Version 5.0 will both support use of Microsoft BASIC .OBJ modules from C programs; they will be available in September. Although Borland offers both Turbo C and Turbo BASIC, .OBJ files from the two compilers can't be combined in a single program. If you have practical experiènce with mixed language BASIC and C programs, share it with Susan and with other readers.

I can't provide a complete answer, but I can identify some potential problems you will face. The two languages may use different segment and group name conventions. Both Lattice and Microsoft C provide compile time options for specifying non- standard segment and group names. Calling conventions may be different but that can be patched with short .ASM glue functions. Another problem is that BASIC and C compilers generate .OBJ modules that may depend on that language's initialization sequence having been executed. To minimize problems, use one language for I/O, dynamic memory allocation and other library functions. When you link the combined C and BASIC program, you may need .LIB files for both languages. Watch out for name conflicts between library functions from the two sets of libraries.

Mailing List processing - James E. Graf (Express Marketing Inc., 2328 Hammond Drive, Schaumberg, IL 60173) is developing a mailing house application. He asks for information about any work done on address parsing mechanisms to reduce mailing addresses to their component parts.

Linking Problem - Both K&R and the ANSI C draft specify that upper and lower case letters are not equivalent. This was a stupid decision but we are stuck with it. But some versions of C compilers, the IBM/Microsoft Macro Assembler and the MS-DOS linker may not distinguish between upper and lower cases in public names. If you don't match up compiler, assembler and linker options correctly, you will get error messages when you try to link C programs. Here are some guides for the programs involved:

The MS-DOS LINK program ignores case differences in public names as the default option. The /NOIGNORECASE option in recent versions of the linker preserves case-sensitivity. Unless you have deliberately used public names that differ only in letter case, let the linker ignore case-sensitivity.

Lattice C normally records all public names as upper case in the .OBJ modules. This avoids bugs in some linker versions but it doesn't allow names such as fred and FRED to be distinguished. In Version 3.2 of Lattice C, the -a option preserves letter case in public names.

The behavior of Microsoft C depends on the command you use. If you use the MSC command and execute LINK separately, case differences in public names are ignored by the linker. But if you use the CL command, case differences are recognized by the linker (CL invokes LINK with the /NOIGNORECASE option specified.)

The Turbo C compiler preserves case sensitivity and the Turbo Linker is sensitive to letter case.

The MASM Macro Assembler normally makes public names upper case in the .OBJ file. In recent versions, the /MX option preserves letter case in public names. The /ML option preserves case sensitivity in all variables and labels. Use the /ML or /MX options with Microsoft C or Turbo C programs.

Lattice and Microsoft C Users' Group
P.O. Box 271965
Concord, CA 94527
USA

First Class
U. S. Postage
PAID
Concord, CA
Permit no. 316

Last Issue 11/86 Member No. 1162

Ted A. Reuss
Information Communication Corp.
6300 Hillcroft, Suite 304
Houston TX 77081

LMCUG Catalog / Order Form

You may copy this page as a quick order form. Mark the items you are ordering and enclose a check for the total amount payable to Lattice and Microsoft C Users' Group. Checks must be in US Dollars and drawn on a US bank or a US branch of a foreign bank. No credit card charges.

__ Address Change - just mark your changes on the mailing label above.

Membership includes 6 bi-monthly issues of the newsletter - $30 per year ($40 in Canada, S.America,Europe and $45 in Asia or Africa.)
__ New Membership - list your mailing address and the compiler you use.
__ Membership renewal - Include member number from mailing label.

Back issues are $7.50 per issue ($9.00 outside the USA.)
__ No 1 - reducing C program size, using the Librarian, Lattice C bugs & fixes, running batch files from C programs,news on vers. 2.15 of Lattice C, sources for technical information on C and the IBM PC.
__ No 2 - keyboard input, dynamic storage allocation, a text sorting program. Functions for data-independent quicksort, accessing BIOS and DOS, cursor sensing & setting, elapsed time, and menu display.
__ No 3 - the ANSI C standard, a review of Microsoft 3.0, Text /Binary I/O modes, coding style, mixing scanf and gets and using assembler language with C.
__ No 4 included reviews of the BRIEF editor and of version 3.00 of Lattice C, a program totaling a list of file sizes, a guide to Lattice C startup code, a program for locating syntax errors, and bug reports.
__ No 5. included a stack usage monitor module, tools for memory resident programs and a review of The C Wizard's Programming Reference.
__ No. 6 - articles on using header files and coping with ANSI C changes, a memory allocation interface, linked list and LRU I/O cache modules.
__ No. 7 - articles on alignment problems, faster buffered I/O, Lattice C large models, forcing file updates, portability in practice I and reviews of MASM 4.0 and Efficient C.
__ No. 8 - reviews of books on algorithms, Lattice 3.1, Microsoft 4.0, articles on code generation, problems with arrays larger than 32k bytes, pointers to functions and source code for printer output and getting command line arguments.
__ No. 9 - reviews of C-Terp and Instant C interpreters, articles on books on MS-DOS programming and functions with a variable number of arguments. Source code for for scanning environment strings and parsing file names.
__ No. 10 - sources for info on memory resident pgms, Lattice and Microsoft bugs, portability in practice II, source code for reading Mail Merge files. Tools for using hash table techniques.
__ No. 11 - code generation, Part 2, Char table functions, Handling fixed length char arrays in C, books on Data Communications, and output to a serial printer with Xon/Xoff protocol.
__ No. 12 -Tools for Portable ASM code, correct pointer differences, finding out how much memory is available, CRC calculation, reading raw disk sectors and a program for verifying floppy disk copies.

Supplementary Disks are $ 15 ($ 18 outside the USA.)

__ SD 5 - source code from issues # 2 and # 5 (stack monitoring and memory resident tools) plus device driver tools and a wild-card expansion function. (Some Lattice C specific code.)
__ SD 9 - source code from issues 1, 3, 4, 6, 7, 8 and 9. GREP pattern matching program.
__ SD 10 - source code from issue 10. Extra material on reading MailMerge files and using hash tables.